Mini Project 3 ECSE 551

Shankhin Brahmavar, Cameron Butterfield
260921778, 260789137
shankhin.brahmavar@mail.mcgill.ca, cameron.butterfield@mail.mcgill.ca

Abstract

One area of problem solving that machine learning plays a very large role in is image understanding and classification. This assignment focused on this use of machine learning to interpret image data. For this assignment each group was tasked with classifying a images from a given dataset into 10 classes. This was performed using a Convolutional Neural Network (CNN). The parameters for this CNN were then fine tuned in order to produce a final model with $\sim 84\%$ accuracy.

1 Introduction

The task in this mini project involved the classification of an image dataset. The images found within the dataset each contained both articles and digits. The images also had an associated article price. The goal was to correctly output the article's price based on the image.

For training samples provided in the training data there was a given image and associated label, making this a supervised learning task. In order to produce a supervised classification model to predict the labels a number of steps were required. First the data was to be loaded, next a neural network was designed, the data was fed into this neural network and the neural network was trained. Once this was completed the network design was tested and adjusted to optimize accuracy.

An important tool which was used at each of these steps was PyTool. Its dataset class and dataloader class were used read the data, input the data into the network and provide and interface to access all the training and testing samples. The Convolutional Neural Network (CNN) that was used was also heavily based off the PyTorch neural network. This was

2 Dataset

The dataset provided was a modified version of the MNIST + Fashion-MNIST datasets constructed for this mini-project. This dataset consisted of 60,000 training images saved in a pkl file. These training images were accompanied by the corresponding target labels save in a csv file. Also included was a test set consisting of 1000 images in pkl file for the purpose of testing the accuracy of the model after training.

The 60,000 training images and corresponding target labels where downloaded and stored in one a dataset. The 60,000 samples were evenly distributed between the 10 different classes as can be seen in figure 1. This dataset was then split into a training set and a validation set. The training set was 48,000 samples and the validation set was 12,000. This allowed for the testing of model performance on a validation set with know target labels. This was used to fin tune hyperparameters in order to optimize the performance of the model while also monitoring for over fitting to the training set.

The dataset was then divided into batches of size 16 in order to perform mini-batch gradient descent. This separates the data into batches of 16 samples to be worked through before the automated update of internal model parameters.

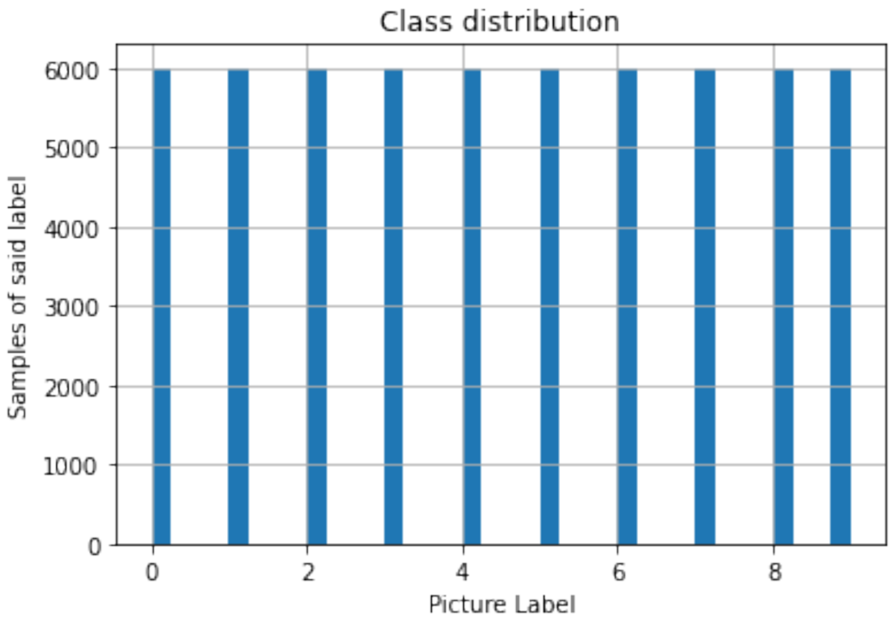


Figure 1: Class distribution of dataset

3 Proposed Approach

The proposed design approach for predicting the target label between 0 and 9 based on the dataset created from the provided images is a convolutional neural network (CNN). This will be implemented based on the PyTorch neural network design provided in tutorial 6.

3.1 Model Design

This neural network structure consists of a series of convolution layers, a max pooling filter, an imaginary layer which restructures the data, and then a series of feed-forward layers. The first series of three convolution layers each have a kernel size of 3. Each of these layers is used in combination with a ReLU activation which simply sets all values less than 2 equal to 2 using a piece-wise function. The output of these first three layers is then passed through a max pool function with a kernel of 2 which essentially halves the size of the information matrix.

After this another similarly designed set of three convolutions and max pool function were used.. Following this an imaginary layer was used to convert the tensor to a vector of 1024 elements.

This was then followed by a series of three feed-forward layers which converted the vector to the final output size of 10. Once this vector was created a log softmax activation was used to output a predicted label.

3.2 Training

In order to optimize the models accuracy with its given architecture a repetitive training process was used. This involved the 16 batches described in Section 2. The model was trained over the dataset repeatedly, continuously updating the internal model parameters after each batch.

In this process, the number of times all 16 datasets have updated the model parameters is described by the hyper-parameter epochs. This was done until the the model completed 100 straight epochs without a significant improvement in the model.

To illustrate this automated parameter adjustment model training process figure 2 was created. This figure displays the negative log likelihood loss (on the y-axis) in relation to the number of epochs complete (on the x-axis). This gives us a visualisation of how the likelihood loss decreased over the

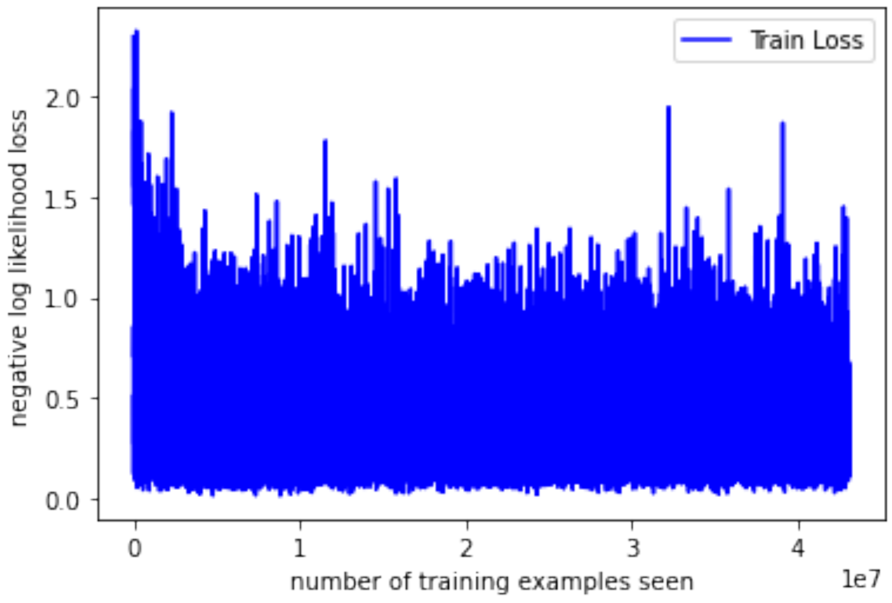


Figure 2: Negative log likelihood of loss vs number of epochs

course of the training process. The graph shows a very rapid improvement in accuracy from zero to about 3 million epochs. After this point, the loss likelihood fluctuates greatly while still producing slightly more effective models, but no clear pattern of improvement.

4 Results

The final model created using this training process was tested throughout training and finally tested after training using the validation set. Using the validation set with known target labels the model could be tested and the predictions could be compared against the known labels to determine the accuracy of the model. Using this testing mechanism it was determined that he model had an accuracy of $\sim 83\%$.

In addition to the overall accuracy a confusion matrix was created to give a break down of the accuracy in predicting images from each class. This confusion matrix shows the percentage of images from each class which were assigned each level. The confusion matrix in figure 3 shows this distribution for our model.

A perfect confusion matrix would show 0 in every position except the diagonal where the values would be 1.00. This distribution shows that the most difficult class to predict was the number 6. This was only correctly labeled in 54% of the samples. This class was incorrectly labeled 0, 2, and 4 each over 10% of the time. Class 2 was also incorrectly labeled 4 in over 10% of cases. The inverse of these incorrect prediction was also true, with classes 0, 2, and 4 all being mislabeled 6 and and class 4 being mislabeled 2 at a higher than average rate.

Finally, the model was tested on a provided test set with unknown target labels. The evaluation of the models performance on this test set was done externally through Kaggle where the desired targets were known. For this test our model performed with an accuracy of 83.766%

5 Discussion and Conclusion

This project helped show the power of CNNs in the classification of images. With the aid of PyTorch we were able to load and process the data as well as come up with a basic architecture for the neural network model itself. With a little fine tuning this model was quickly able to predict the output data at a success rate of $\sim 84\%$, a reasonably successful benchmark for an evenly distributed 10 class problem.

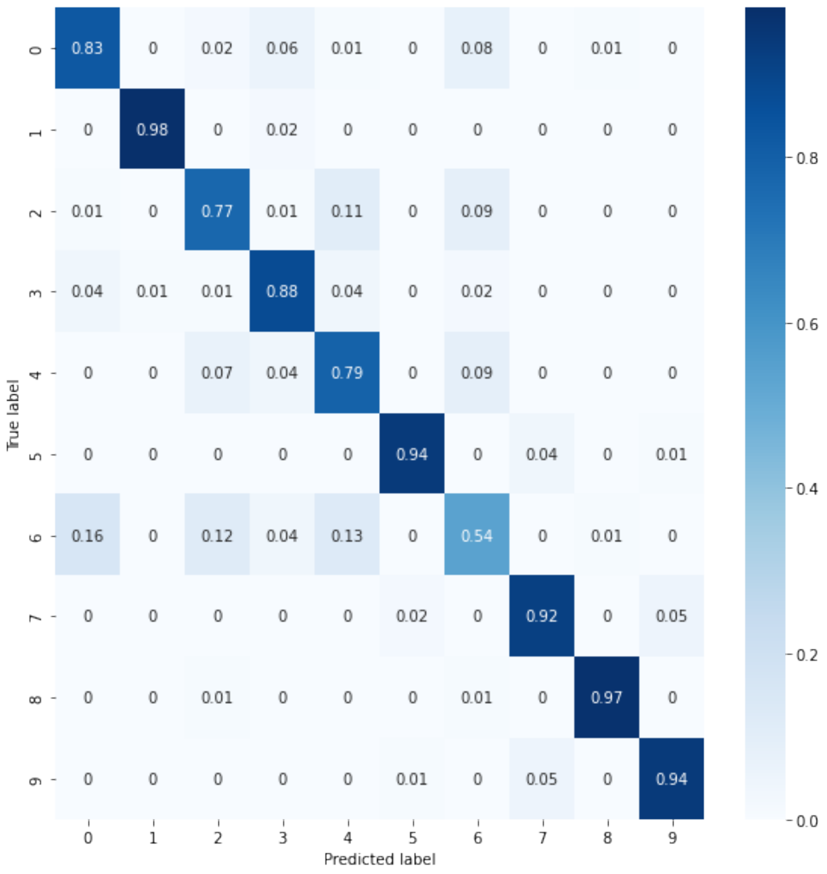


Figure 3: Negative log likelihood of loss vs number of epochs

One characteristic of this neural network model that was examined during the project was its loss likelihood in relation to the number of epochs completed. This comparison showed that beyond a certain epoch level, while improvements still occurred they were few and far between, while the most drastic improvements to the accuracy of the model occurred during the first 3 million epochs. This shows that the rate of improvement decreases as the number of epochs increases.

A technique that could be used to improve the model would be to consider regularization. This would involve penalizing the weight matrices of nodes which are producing inaccurate predictions. This technique can help reduce the effect of over fitting.

One other way to improve the model in this specific case would be to separate the original image into 4 segments. Since the data contained in the image was separated into four quadrants, this would allow the model to analyze each segment individually without the pieces of information interfering with each other. Once the quadrants had been interpreted the model could combine these outputs to produce a more accurate final prediction.

6 Statement of Contributions

For the division of work we split the total project by the two main tasks, with the idea planning and review/finalising of each part being done together. Shankhin was responsible for the data processing and model design that were completed using python. Cameron was responsible for writing the report.

7 Appendix

Import libraries

```
import pickle
import pandas as pd
import seaborn as sns
import tensorflow as tf
import matplotlib.pyplot as plt
import numpy as np
from torchvision import transforms
from torch.utils.data import Dataset
from torch.utils.data import DataLoader, TensorDataset
from PIL import Image
from google.colab import drive
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
import torch.optim.lr_scheduler as scheduler
import datetime
import IPython
device = torch.device('cuda:0' if torch.cuda.is_available() else 'cpu')
drive.mount('/content/gdrive')
```

Download test and train files

```
!pip install kaggle --upgrade -q
%env KAGGLE_USERNAME=shankhinbrahmavar
%env KAGGLE_KEY=aefdd0d58d831be445ba8ebcf91bf723
!kaggle competitions download -c ecse-551-f21-mini-project-3
```

```
env: KAGGLE_USERNAME=shankhinbrahmavar
env: KAGGLE_KEY=aefdd0d58d831be445ba8ebcf91bf723
Warning: Looks like you're using an outdated API Version, please consider updati
Downloading ExampleSubmissionRandom.csv to /content
  0% 0.00/77.1k [00:00<?, ?B/s]
100% 77.1k/77.1k [00:00<00:00, 69.9MB/s]
Downloading Test.pkl.zip to /content
100% 4.03M/4.03M [00:00<00:00, 41.1MB/s]

Downloading Train_labels.csv to /content
  0% 0.00/517k [00:00<?, ?B/s]
100% 517k/517k [00:00<00:00, 160MB/s]
Downloading ReadMe.txt to /content
  0% 0.00/517 [00:00<?, ?B/s]
100% 517/517 [00:00<00:00, 453kB/s]
Downloading LoadData.ipynb to /content
  0% 0.00/49.9k [00:00<?, ?B/s]
```

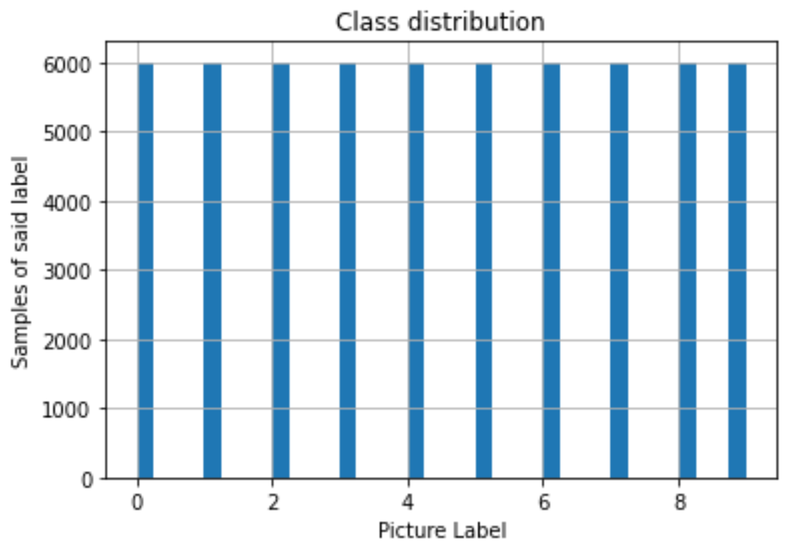
```
100% 49.9k/49.9k [00:00<00:00, 51.2MB/s]
Downloading Train.pkl.zip to /content
 37% 9.00M/24.1M [00:00<00:00, 33.9MB/s]
100% 24.1M/24.1M [00:00<00:00, 61.0MB/s]
```

```
!unzip -o -q '*.zip'
!rm *.zip
train_images = pickle.load(open('/content/Train.pkl', 'rb'))
train_labels = np.genfromtxt('/content/Train_labels.csv', delimiter=',', skip_heade
train_labels = train_labels[np.logical_not(np.isnan(train_labels))]
test_images = pickle.load(open('/content/Test.pkl', 'rb'))
```

```
2 archives were successfully processed.
```

Visualize data

```
plt.hist(train_labels,·bins='auto')
plt.title("Class·distribution")
plt.ylabel("Samples·of·said·label")
plt.xlabel("Picture·Label")
plt.grid()
plt.show()
```



Implement transformation and dataset function

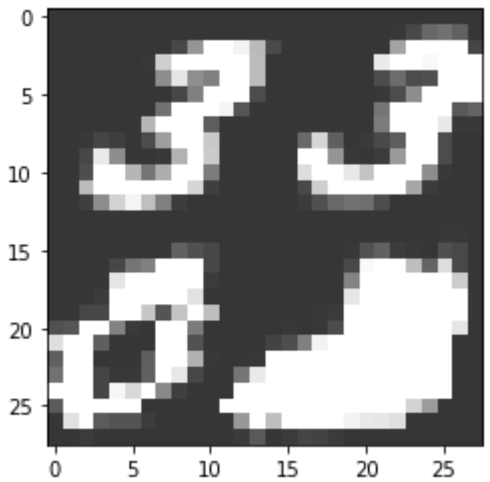
```
# train_images1 = np.where(train_images >= 0.3, 1.0, 0.0)
mean, std = np.mean(train_images), np.std(train_images)
image_transform = transforms.Compose([
                     transforms.ToTensor(),
                     transforms.Normalize((mean,), (std,)),
                  transforms.RandomAffine(degrees = 15, shear = 20, fill = -0.5
              ])

class MyDataset(Dataset):
    def __init__(self, img_file, label_file, transform=None):
        self.data = img_file
        self.targets = label_file
        self.transform = transform

    def __len__(self):
        return len(self.targets)

    def __getitem__(self, index):
        img, target = np.expand_dims(np.squeeze(self.data[index]), axis = 2), int(s
        # img = np.where(img >= 0.5, 1.0, 0.0)
        if self.transform is not None:
           img = self.transform(img)
        img = img.float()
        img = img.to(device)
        target = torch.tensor(target).to(device)
        return img, target
```

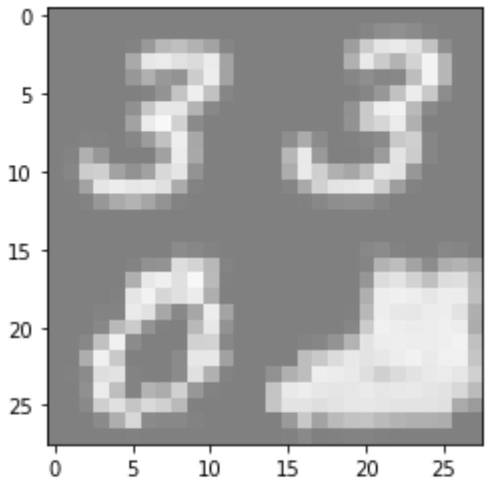
Split train data into training and validation data

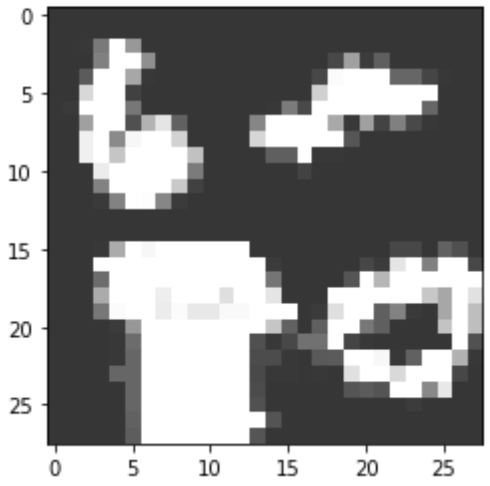
```
dataset = MyDataset(train_images, train_labels, transform = image_transform)
train_dataset, test_dataset = torch.utils.data.random_split(dataset,[48000,12000])
```

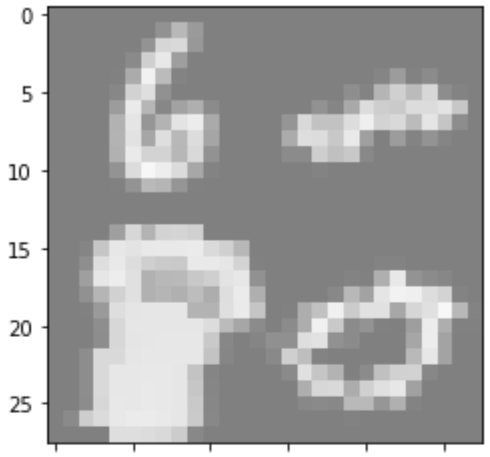
Display a set of images

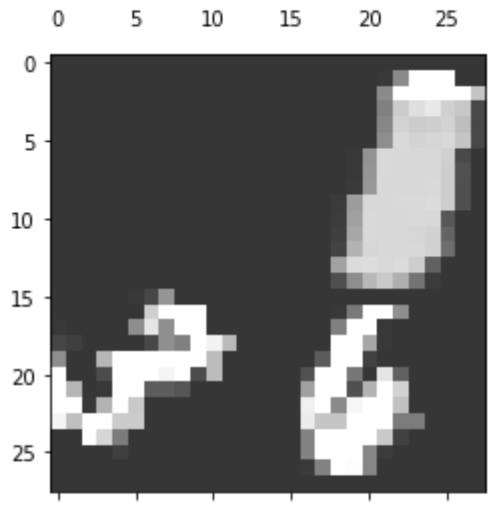
```
def show_image(imgs):
    plt.imshow(imgs.cpu().numpy(), cmap='gray', vmin=-1, vmax=1) #.transpose()
    plt.show()

for i in range(10):
  imgs, labels = dataset[i]
  imgs = np.squeeze(imgs)
  imgs1 = train_images[i]
  imgs1 = np.squeeze(imgs1)
  show_image(imgs)
  plt.imshow(imgs1, cmap='gray', vmin=-1, vmax=1) #.transpose()
  plt.show()
```

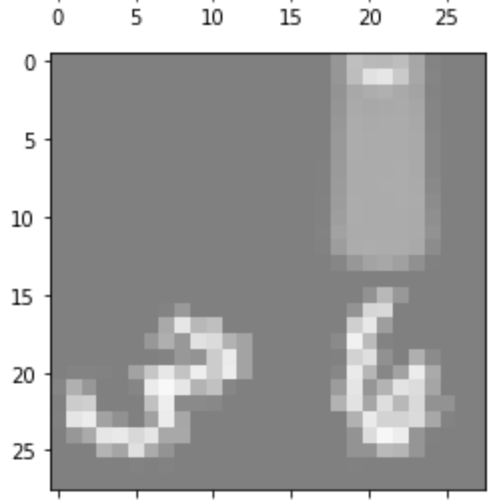

0
5
10
15
20
25
0
5
10
15
20
25

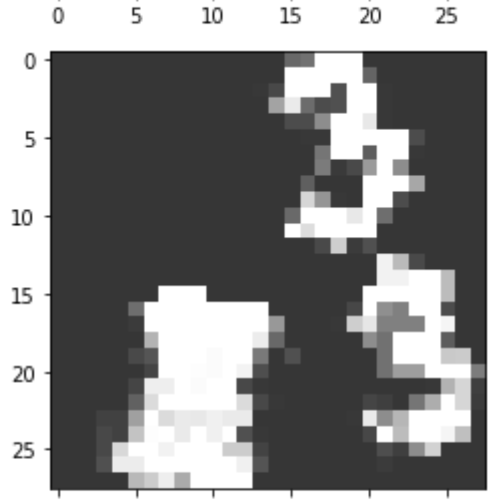

0
5
10
15
20
25
0
5
10
15
20
25

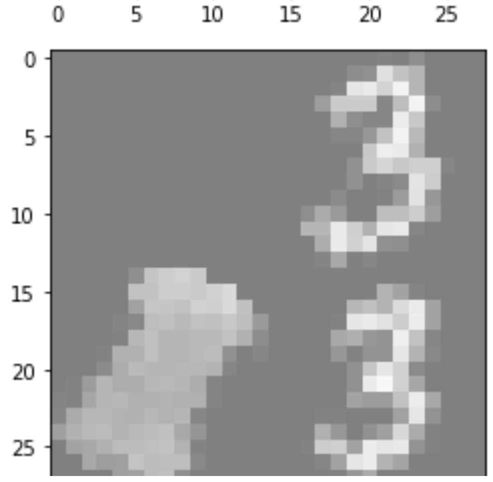

0
5
10
15
20
25
0
5
10
15
20
25

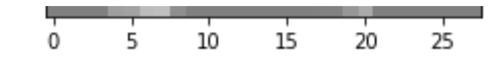

0
5
10
15
20
25

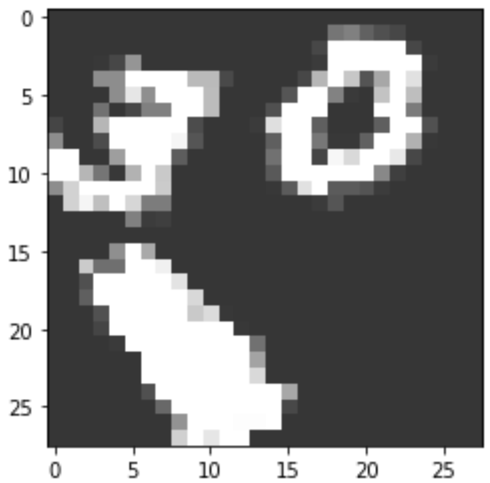

0
5
10
15
20
25
0
5
10
15
20
25

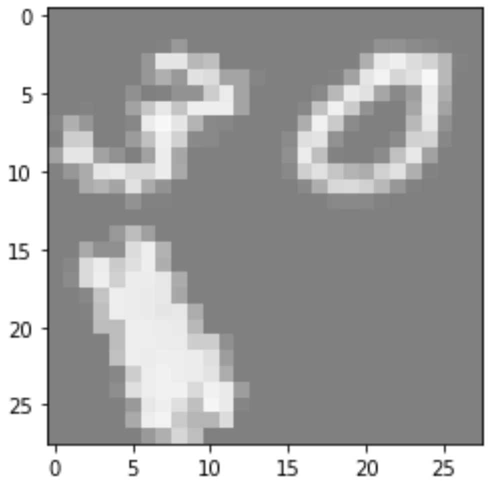

0
5
10
15
20
25
0
5
10
15
20
25

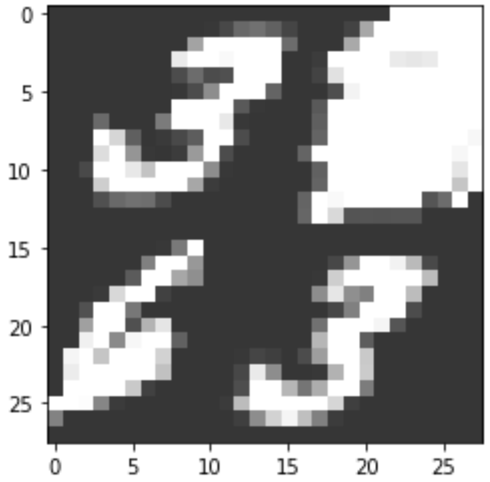

0
5
10
15
20
25
0
5
10
15
20
25

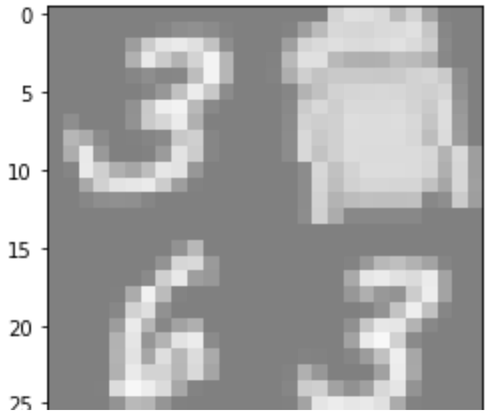

0
5
10
15
20
25

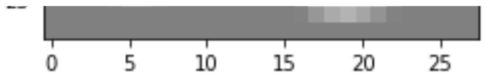


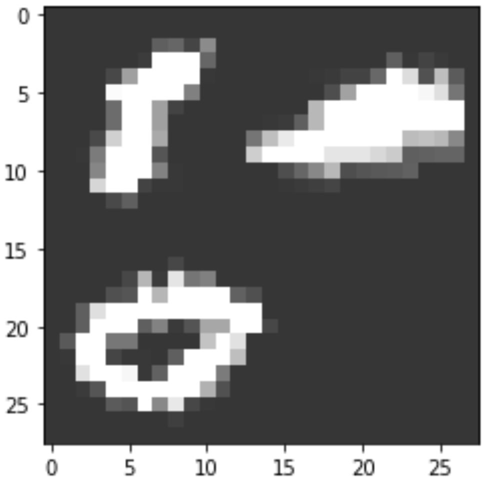


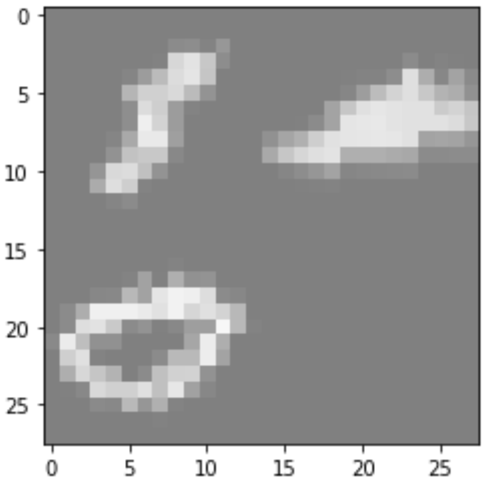


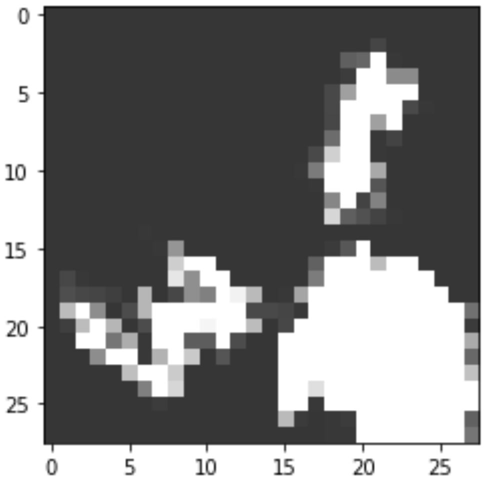


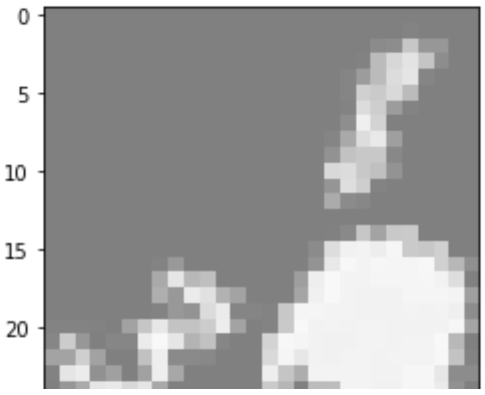












Divide dataset into batches using Dataloader

```
train_loader = DataLoader(train_dataset, batch_size = 16, shuffle=True)
test_loader = DataLoader(test_dataset, batch_size = 16, shuffle=True)
```

Define neural network class

```
class Net(nn.Module):
  def __init__(self):
        super(Net, self).__init__()
        # At first there is only 1 channel (greyscale). The next channel size will
        self.conv1 = nn.Conv2d(1, 32, kernel_size=3)
        # Then, going from channel size (or feature size) 10 to 20.
        self.conv2 = nn.Conv2d(32, 32, kernel_size=3)
        self.conv3 = nn.Conv2d(32, 64, kernel_size=3)
        self.m1 = nn.BatchNorm2d(64)

        self.conv4 = nn.Conv2d(64, 128, kernel_size=3)
        self.conv5 = nn.Conv2d(128, 128, kernel_size=3)
        self.conv6 = nn.Conv2d(128, 256, kernel_size=3)
        self.m2 = nn.BatchNorm2d(256)

        # Now let us create some feed foreward layers in the end. Remember the size
        self.fc1 = nn.Linear(1024, 512)
        # The last layer should have an output with the same dimension as the numbe
        self.fc2 = nn.Linear(512, 256)
        self.fc3 = nn.Linear(256, 10)
        self.dropout = nn.Dropout(0.25)


    # And this part defines the way they are connected to each other
    # (In reality, it is our foreward pass)
  def forward(self, x):


      # F.relu is ReLU activation. F.max_pool2d is a max pooling layer with n=2
      # Max pooling simply selects the maximum value of each square of size n. Effe
      # At first, x is out input, so it is 1x28x28
      # After the first convolution, it is 10x24x24 (24=28-5+1, 10 comes from featu
      # After max pooling, it is 10x12x12
      # ReLU doesn't change the size

      x = F.relu(self.conv1(x), 2)
      x = F.relu(self.conv2(x), 2)
      x = F.relu(F.max_pool2d(self.conv3(x), 2))
      x = self.m1(x)
```

```

        # Again, after convolution layer, size is 20x8x8 (8=12-5+1, 20 comes from fea
        # After max pooling it becomes 20x4x4

        x = F.relu(self.conv4(x), 2)
        x = F.relu(self.conv5(x), 2)
        x = F.relu(F.max_pool2d(self.conv6(x), 2))
        x = self.m2(x)

        # This layer is an imaginary one. It simply states that we should see each me
        # as a vector of 320 elements, instead of a tensor of 20x4x4 (Notice that 20
        x = x.view(-1, 1024)

        # Feedforeward layers. Remember that fc1 is a layer that goes from 320 to 50
        x = F.relu(self.fc1(x))
        x = self.dropout(x)

        # Output layer
        x = F.relu(self.fc2(x))
        x = self.dropout(x)

        x = self.fc3(x)

        # We should put an appropriate activation for the output layer.
        return F.log_softmax(x)
```

Define optimization function and learning rate decay function

```
network = Net()
network.to(device)

optimizer = optim.AdamW(network.parameters(), lr = 0.001)
lr_scheduler = scheduler.ReduceLROnPlateau(optimizer)

train_losses = []
train_counter = []
test_losses = []
test_counter = [i*len(train_loader.dataset) for i in range(3)]
```

Define train and test functions

```
def train(epoch):
  network.train()
  for batch_idx, (data, target) in enumerate(train_loader):
    optimizer.zero_grad()
    output = network(data)
    loss = F.nll_loss(output, target) #negative log liklhood loss
    loss.backward()
    optimizer.step()
    if batch_idx % 50 == 0:
      out.update(IPython.display.Pretty('Train Epoch: {} [{}/{} ({:.0f}%)]\tLoss: 
        epoch, batch_idx * len(data), len(train_loader.dataset),
        100. * batch_idx / len(train_loader), loss.item())))
      train_losses.append(loss.item())
      train_counter.append(
        (batch_idx*64) + ((epoch-1)*len(train_loader.dataset)))

path = '/content/gdrive/MyDrive/Colab Notebooks/Project 3/'
model_name = 'model1.pth'
def test(best_loss):
  network.eval()
  test_loss = 0
  correct = 0
  with torch.no_grad():
    for data, target in test_loader:
      output = network(data)
      test_loss += F.nll_loss(output, target, size_average=False).item()
      pred = output.data.max(1, keepdim=True)[1]
      correct += pred.eq(target.data.view_as(pred)).sum()
  test_loss /= len(test_loader.dataset)
  test_losses.append(test_loss)
  print('\nTest set: Avg. loss: {:.4f}, Accuracy: {}/{} ({:.0f}%)'.format(
    test_loss, correct, len(test_loader.dataset),
    100. * correct / len(test_loader.dataset)))
  if test_loss < best_loss or best_loss == 0:
    print('Saving new best model to {} \n'.format(path))
    torch.save(network.state_dict(), '{}{}'.format(path, model_name))
    torch.save(optimizer.state_dict(), '{}optimizer.pth'.format(path))
    return test_loss
  return best_loss
```

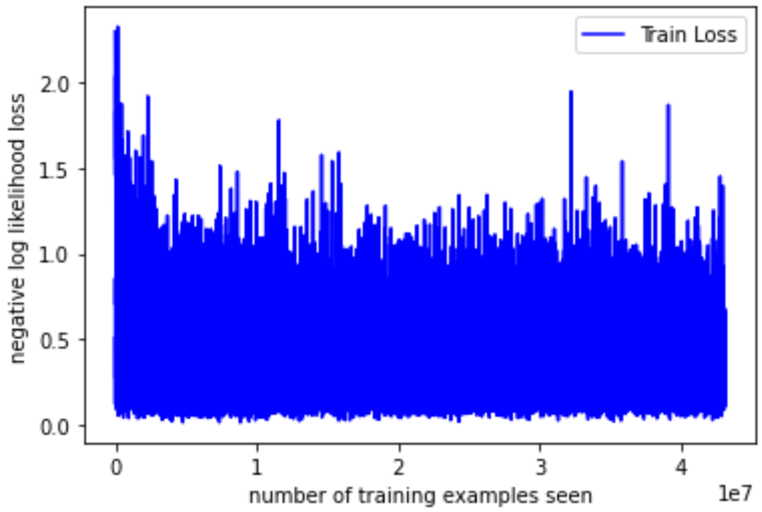
Train until loss doesn't change over an interval

```
print('Start training:', datetime.datetime.now())
patience = 100
counter = 0
epoch = 0
best_loss = 0

while(counter <= patience):
  out = display(IPython.display.Pretty(''), display_id=True)
  train(epoch)
  new_loss = test(best_loss)
  if new_loss <= best_loss or best_loss == 0:
    counter = 0
    best_loss = new_loss
```

```
else:
    counter += 1
lr_scheduler.step(new_loss)
epoch += 1
```

```
fig·=·plt.figure()
plt.plot(train_counter,·train_losses,·color='blue')
plt.legend(['Train·Loss'],·loc='upper·right')
plt.xlabel('number·of·training·examples·seen')
plt.ylabel('negative·log·likelihood·loss')

```

```
Text(0, 0.5, 'negative log likelihood loss')
```



Prediction on test data

```
    8   out = display(IPython.display.Pretty(''), display_id=True)
```

```
class testDataset(Dataset):
    def __init__(self, img_file, transform=None):
        self.data = img_file
        self.transform = transform

    def __len__(self):
        return len(self.data)

    def __getitem__(self, index):
        img = np.expand_dims(np.squeeze(self.data[index]), axis = 2)
        # img = np.where(img >= 0.5, 1.0, 0.0)
        if self.transform is not None:
           img = self.transform(img)
        img = img.float()
        img = img.to(device)
        return img
```

```
testData = testDataset(test_images, transform = image_transform)
test_set_loader = DataLoader(testData, batch_size = 10000, shuffle=False)
```

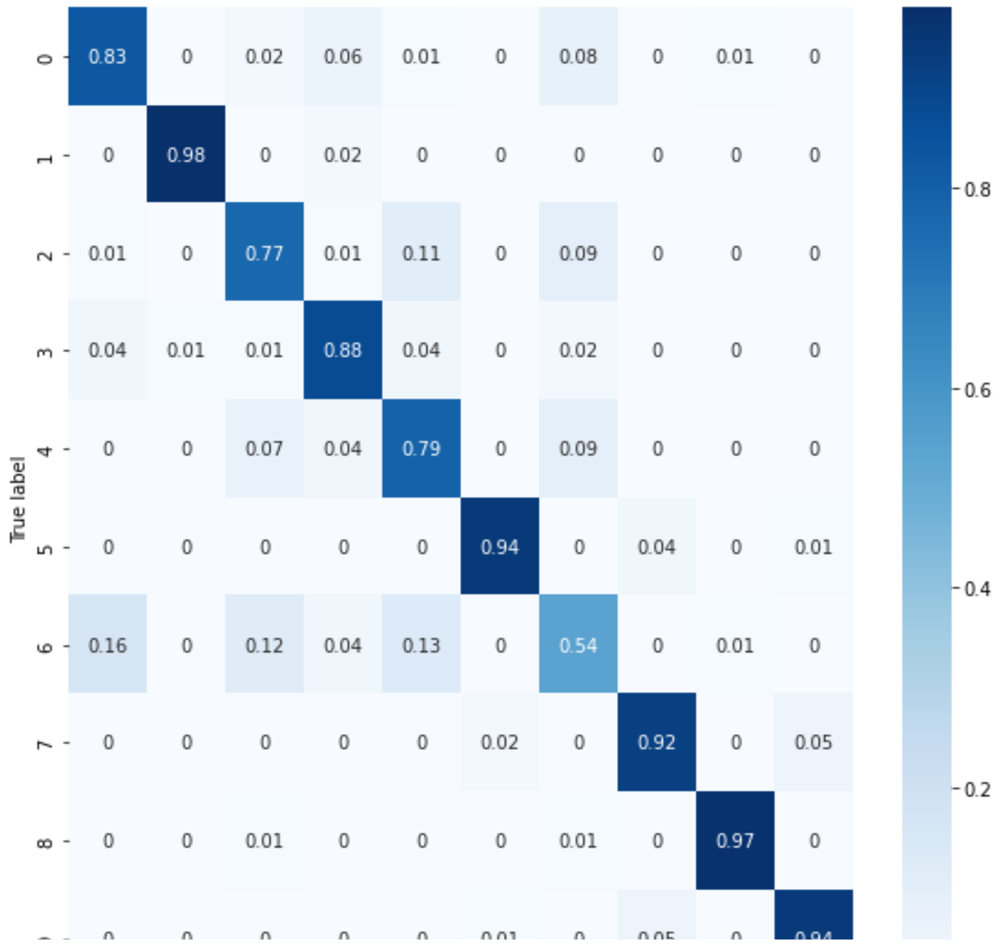
```
print("Evaluate on test data")

network = Net()
```

```
network.load_state_dict(torch.load('/content/gdrive/MyDrive/Colab Notebooks/Project
network.eval()
with torch.no_grad():
  for data in test_set_loader:
    output = network(data.cpu())
    results = output.data.max(1, keepdim=True)[1]

results = results.cpu()
results = np.squeeze(results.numpy())
print(np.shape(results))
dataframe = {'id':np.arange(len(results)),'class':results}
df = pd.DataFrame(dataframe)
df.to_csv('results.csv',index=False)
```

```
Evaluate on test data
(10000,)
/usr/local/lib/python3.7/dist-packages/ipykernel_launcher.py:64: UserWarning: Im
```

```
train_loader = DataLoader(train_dataset, batch_size = 48000, shuffle=True)
```

```
# Confusion matrix

# network = Net()
# network.load_state_dict(torch.load('/content/gdrive/MyDrive/Colab Notebooks/Proje
# network.eval()
# with torch.no_grad():
#   for data, label in train_loader:
#     output = network(data.cpu())
#     labels = label
#     results = output.data.max(1, keepdim=True)[1]

classes = [0,1,2,3,4,5,6,7,8,9]

con_mat = tf.math.confusion_matrix(labels=labels.cpu(), predictions=results.cpu(),
con_mat_norm = np.around(con_mat.astype('float') / con_mat.sum(axis=1)[:, np.newaxi

con_mat_df = pd.DataFrame(con_mat_norm,
                     index = classes,
                     columns = classes)

figure = plt.figure(figsize=(8, 8))
sns.heatmap(con_mat_df, annot=True,cmap=plt.cm.Blues)
plt.tight_layout()
plt.ylabel('True label')
plt.xlabel('Predicted label')
plt.show()
```


True label
0
1
2
3
4
5
6
7
8
0.8
0.6
0.4
0.2
0.83 0 0.02 0.06 0.01 0 0.08 0 0.01 0
0 0.98 0 0.02 0 0 0 0 0 0
0.01 0 0.77 0.01 0.11 0 0.09 0 0 0
0.04 0.01 0.01 0.88 0.04 0 0.02 0 0 0
0 0 0.07 0.04 0.79 0 0.09 0 0 0
0 0 0 0 0 0.94 0 0.04 0 0.01
0.16 0 0.12 0.04 0.13 0 0.54 0 0.01 0
0 0 0 0 0 0.02 0 0.92 0 0.05
0 0 0.01 0 0 0 0.01 0 0.97 0